4. Алгоритми и програми

Проф. д-р Емил Хаджиколев

Алгоритъм и програма

- **Алгоритъм е описание на последователност от краен брой стъпки** (действия, инструкции)**, които се изпълняват за решаването на задача или клас от задачи**.
  - Задачите може да са математически или от друга предметна област.
- **Програма е алгоритъм, който може да бъде изпълнен от компютър** (или друго устройство)**.**
  - За да се изпълни програма написана на език от високо ниво, тя се превежда до инструкции на машинен език, с помощта на транслиращи програми.

Примери за алгоритми

- Примерни задачи, за които може да се опише алгоритъм (не задължително като програма):
  - намирането на корените на квадратно уравнение ($ax^2 + bx + c = 0$) по формула $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$;
  - рецепта за приготвяне на ястие;
  - инструкция за сглобяване на шкаф;
  - и др.

Алгоритъм за изчисляване на периметър и лице на правоъгълник по зададени дължини на две прилежащи страни

1. Въвеждане дължината **a** на едната страна на правоъгълника.
2. Въвеждане дължината **b** на втората страна на правоъгълника.
3. Изчисляване P = 2 * (a + b).
4. Изчисляване S = a * b.
5. Извеждане стойността на P.
6. Извеждане стойността на S.
7. Край.

- За една задача може да се измислят различни решения (алгоритми).
- Една програма реализира един модел за решението на задача.
  - Създаденият модел (програма) зависи от знанията на програмистите, поставени специфични изисквания, възможностите на избрания език за програмиране и др.
- Използването на абстракции/обобщения (например, променливи **a** и **b** в предходния слайд) прави алгоритъма приложим върху множество конкретни данни.

Видове данни в алгоритмите

- Алгоритмите се прилагат върху конкретни **входни данни**.
- По време на изпълнение на алгоритъма обикновено се получават и обработват **междинни данни**.
- **Изходни данни** се получават в резултат от цялостното изпълнение на алгоритъма.

Данни и алгоритми

- Данните могат да бъдат организирани по различни начини:
  - Може да са: числа, низове и от други примитивни или съставни типове; колекция (списък, множество или др.) с известен или неизвестен брой елементи;
  - Може да се четат от файлове или бази от данни, или да се въвеждат от потребител по време на работа на програмата.
- Съответно, **в зависимост от направените избори за структуриране и съхранение на данните**, за решението на една задача може да се съставят много алгоритми и различни конкретни реализации.

Под алгоритми

- В много от случаите част от даден алгоритъм може да бъде обособена като самостоятелен под алгоритъм, който:
  - се ползва многократно в основния алгоритъм;
  - има сложна логика;
  - може да бъде използван и при решения на други конкретни задачи.
- Всеки под алгоритъм има собствени входни данни, междинни и изходни данни.
  - Входните данни се получават от извикващия алгоритъм, а изходните се връщат към него.
- В примера с правоъгълниците, под алгоритми може да бъдат изчисленията на лицето и периметъра.
  - При тях логиката не е много сложна, но при други задачи в един под-алгоритъм може да има сложни изчисления. От друга страна формулите могат да се ползват от други програми, ако бъдат изнесени в отделна библиотека.

Начини за описание на алгоритмите

- **Словесно** – на естествен език;
- **Блок-схема (flow-chart);**
- **Език за програмиране.**

Словесно описание на алгоритмите

- Словесното описание е важна част от документацията на едно приложение и в частност на алгоритмите в него.
- Недостатъци:
  - възможно двусмислено тълкуване на някои езикови конструкции;
  - не добра нагледност;
  - невъзможност да бъдат изпълнени от машина.

Заб.: Съвременните чат-ботове, все по-добре се справят с двусмислията в естествените езици, като използват придобития при обучението си опит и контекста на текущия разговор. Все пак те не задават уточняващи въпроси, а се опитват да предполагат какво се опитва да каже потребителя.

Блок-схемен език

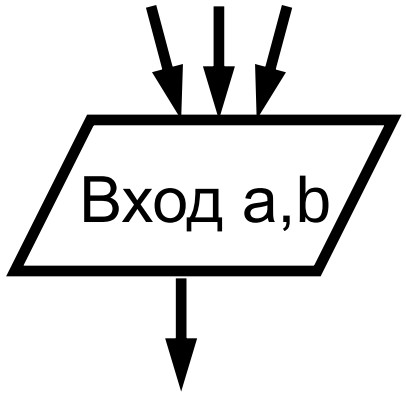
- **Основни елементи в блок-схемите** са:
  - **блокове** за означаването на отделни стъпки;
  - **стрелки**, за указване на последователността на изпълнение на действията.
- Блок-схемите са **удобни за нагледно представяне на алгоритми** при проектиране и в научни публикации.
- **Не са подходящи за изпълнение от компютър**.

Съвместно с други диаграмни езици и нотации (като UML, BPMN, Entity Relationship и др.), блок-схемите се използват в процеса на моделиране на различни елементи в софтуерните приложения: архитектури, процеси, същности/обекти и връзки между тях, алгоритми и др.

Основни блокове в блок-схемите (1)

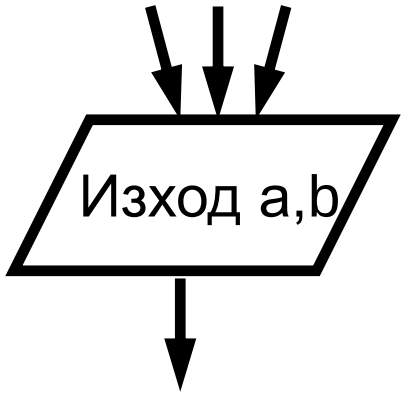
- Блок за **начало** – има форма на елипса, от която излиза само една стрелка.

- Блок за **вход** (успоредник) – в него се записват имената на входни променливи/параметри, които се въвеждат по време на изпълнение на програмата; към блока могат да сочат повече от една стрелка, но излиза само една.

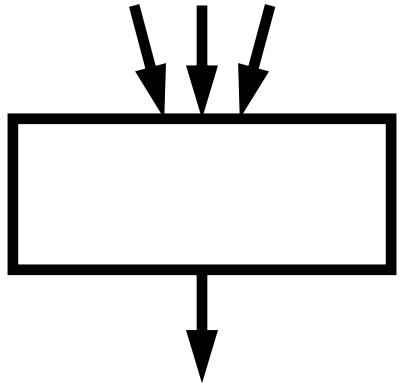


Основни блокове в блок-схемите (2)

- Блок за **изход** (също успоредник) – съдържа имената на изходни променливи; към него могат да сочат повече от една стрелка, но излиза само една.

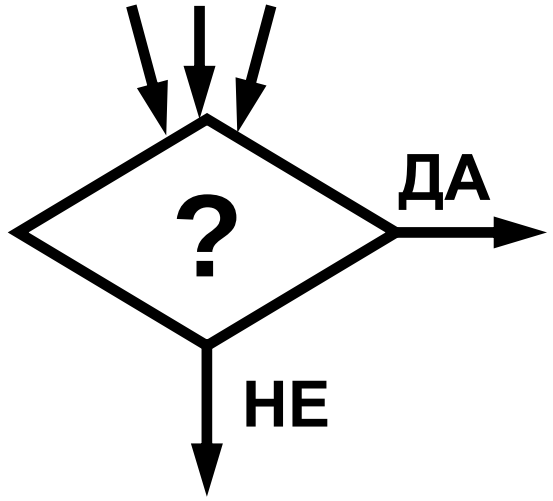


- Блок за **елементарно действие** (правоъгълник) – в него се изписва елементарното действие; към този блок сочи поне една стрелка, а от него излиза винаги само една стрелка.



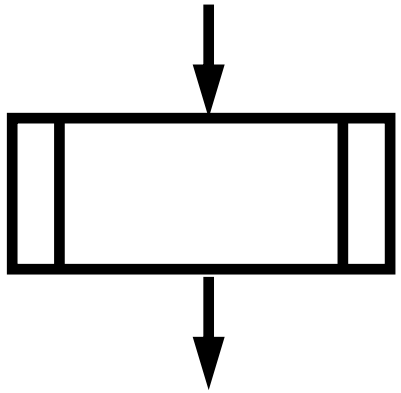
Основни блокове в блок-схемите (3)

- Блок с **логическо условие** (ромб) – в него влиза поне една стрелка и винаги излизат две стрелки; над едната от излизащите стрелки се записва „да“ и тя сочи блока, към който ще продължи изпълнението на алгоритъма, ако логическото условие е вярно. Над другата стрелка, която излиза от блока се записва „не” и тя сочи към блока, който да се изпълни, ако логическото условие е невярно.



Основни блокове в блок-схемите (4)

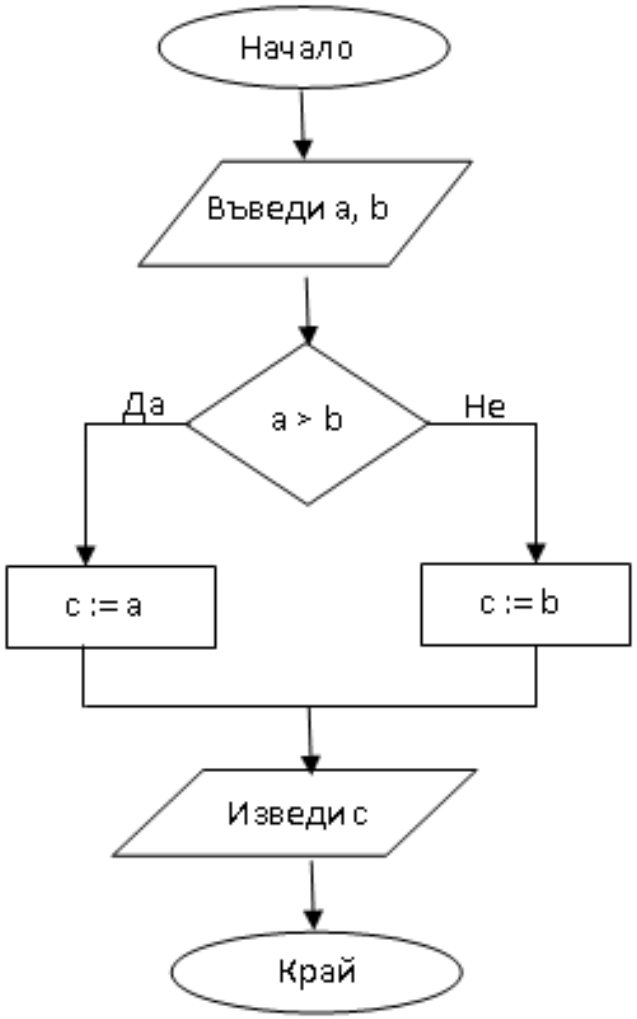
- Блок за **обръщение към под алгоритъм** – тези блокове съдържат име на под алгоритъм (отделна блок-схема).

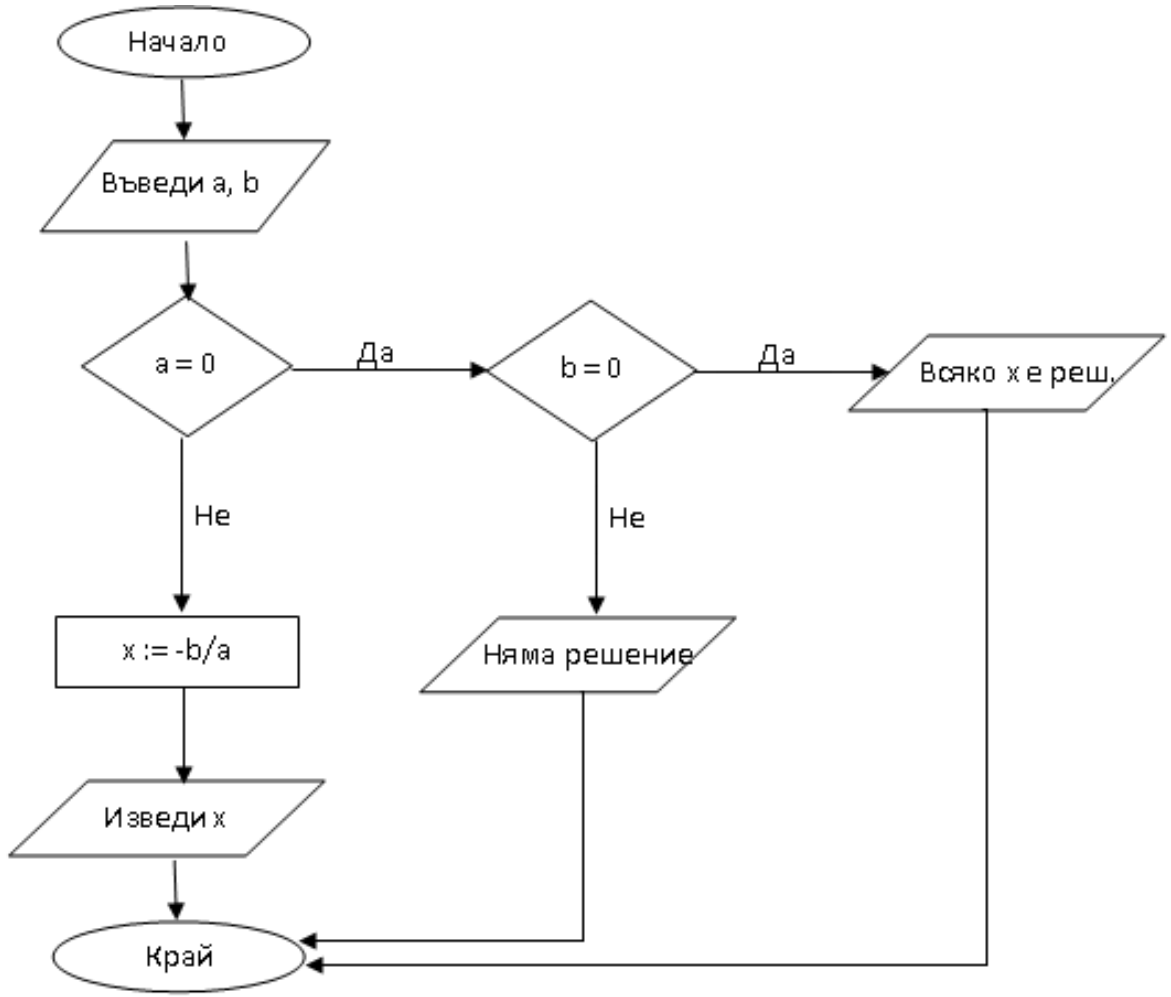


- Блок за **край** (елипса) – във всяка блок-схема има само един блок за край. Към този блок сочат една или повече стрелки, а от него не излиза нито една.

Примерни блок-схеми

(без разяснения)





Програмен език за описание на алгоритмите

Описанието на алгоритъм на ЕП е най-високото ниво на формализация на алгоритмите, даващи възможност да бъдат изпълнени на компютър. За да може да се опише алгоритъм на какъвто и да е ЕП, трябва добре **да се познават**:

- **синтактичните конструкции на езика**;
- **стандартни (и съответно ефективни) решения на основни задачи**;
- **библиотеки**, свързани с решението на задачата – напр., може да е необходимо да се визуализира елементарен алгоритъм чрез сложни графични библиотеки, които може да бъдат използвани наготово;
- **специализирани техники** за разработка – напр. **шаблони за дизайн**;
- и др.

Основни типове алгоритми и езикови конструкции в езиците за програмиране

- **Линейни** – описаните в кода инструкции се изпълняват последователно;
- **Разклонени** – if, if-else, switch-case;
- **Циклични/итеративни** – for, do-while, while, for-each;
- **Рекурсивни** – в кода на подпрограма се извиква същата подпрограма (обикновено) с други параметри.

- Конструкциите са с почти еднакъв синтаксис в съвременните езици за програмиране.

Линейни алгоритми

- **Всички описани стъпки се изпълняват последователно.**
- Описаният по-горе алгоритъм за периметър и лице на правоъгълник е линеен.

Разклонени алгоритми

- В **зависимост от дадени условия** (определени върху входни данни и междинни резултати) **в алгоритъма се предвиждат различни възможни пътища**.
  - При това (очевидно) някои от стъпките ще се пропуснат при изпълнението върху конкретни данни.

Разклонен алгоритъм: Определяне на по-голямото от две числа a и b

1. Въвеждане стойността на **a**.
2. Въвеждане стойността на **b**.
3. **Проверка дали a е по-голямо от b. Ако a е по-голямо от b, преминаване на стъпка 4. В противен случай преминаване на стъпка 5.**
4. Извеждане стойността на a. Преминаване на стъпка 8.
5. **Проверка дали a е по-малко от b. Ако a е по-малко от b, преминаване на стъпка 6. В противен случай преминаване на стъпка 7.**
6. Извеждане стойността на b. Преминаване на стъпка 8.
7. Извеждане на съобщение „Двете числа са равни”.
8. Край.

В този пример, в стъпки 3 и 5 има условия, които определят различни начини, по които може да продължи изпълнението на алгоритъма.

Циклични/итеративни алгоритми

- При тях **част от стъпките на алгоритъма се изпълняват многократно**, обикновено върху различни данни (стойности на участващите променливи).
- Броят на изпълненията се контролира от условие, зависещо от параметри.

Цикличен алгоритъм: сума на числата от 1 до n (n>0)

1. Въвеждане стойността на числото n.
2. Проверка дали n > 0. Ако n > 0, преминаване към стъпка 3, в противен случай преминаване към стъпка 1.
3. Присвояване стойност 1 на променлива за брояч i.
4. Присвояване стойност 0 на променливата за сума S.
5. **Проверка дали i < n. Ако i < n, преминаване на стъпка 6. В противен случай, преминаване към стъпка 8.**
6. **Изчисляване S = S + i.**
7. **Изчисляване i = i + 1. Преминаване към стъпка 5.**
8. Извеждане стойността на S.
9. Край.

Стъпки 5, 6 и 7 може да се изпълняват многократно.

Рекурсивни алгоритми

- **Рекурсивният алгоритъм** е такъв, който на определена стъпка **извиква същия алгоритъм** като под алгоритъм (обикновено с различни входни параметри).
- В него има логическо условие за край, което определя приключването на рекурсивните изпълнения.
- При приключване на рекурсията, първи приключва последно извикания под алгоритъм, след това приключва предпоследния и т.н., последен приключва основния алгоритъм.

Видове рекурсия

- **Директна** – алгоритъма извиква себе си;
- **Индиректна** – един алгоритъм вика друг алгоритъм (с друга логика), който от своя страна се обръща към първия.

Рекурсивни и итеративни решения (Видове алгоритми според реализацията на повтарящите се стъпки)

- **За всеки рекурсивен алгоритъм може да се направи съответен итеративен.**
- Предимства на рекурсивните алгоритми: написания код е по-елегантен, по-кратък (обикновено).
- Недостатъци на рекурсивните алгоритми :
  - заемат повече памет, защото в паметта се пазят данни за всички изпълняващи се подпрограми.
  - по-бавни са (по-същите причини – но зависи от конкретните реализации);
- Итеративните алгоритми са по-ефективни от рекурсивните.

Рекурсивен алгоритъм за намиране на факториел

- Тривиален пример за рекурсивен алгоритъм е намирането на факториел от цяло неотрицателно число n.
- По дефиниция факториел от n се записва с формулата

  n!=n*(n-1)*(n-2)...3*2*1, а

  0! = 1
- Тази дефиниция може да я представим и като рекурсивна

  n!=n*(n-1)!,

  0! = 1
- т.е. за да намерим факториел от n трябва да умножим n с факториел от (n-1). При рекурсивните извиквания входният параметър ще намалява непрекъснато. Условието за спиране на рекурсивните извиквания е входният параметър да е 0 или 1.
- Тази задача може да се реализира и чрез цикличен алгоритъм.

Алгоритъм на Евклид за намиране на най-големия общ делител на две естествени числа n и m

Един от най-древните математически алгоритми.

Не много подробно описани стъпки:

1. Ако m<n, разменяме числата.
2. На m присвояваме нова стойност: остатъка от делението на m с n.
3. Ако m е различно от 0 изпълняваме отново стъпка 2 с новите стойности на числата.
4. (Иначе се получава) резултат: n е търсения най-голям общ делител.

Видове алгоритми според начина на обработка на входните данни

- **Детерминирани** – при всяко изпълнение на алгоритъма върху едни и същи входни данни се изпълняват едни и същи стъпки и се получава един и същи краен резултат.
- **Стохастични** (недетерминирани) – при тях има елемент на случайност и/или евристика, поради което крайните резултати може да се различават при едни и същи входни данни.
  - Използват се при моделиране на ИИ, симулации, за бързо намиране на приблизителни решения и др.

Видове алгоритми според начина на изпълнение на стъпките

- **Серийни** – стъпките се изпълняват последователно.
  - Подходящи за задачи, при които скоростта на изпълнение не е от значение и за компютри с един процесор.
- **Паралелни** – при тях някои части от задачата се решават паралелно и независимо една от друга, с цел по-ефективно използване на ресурсите при многоядрени процесори.
  - Удачно е да се използват само ако основната задача може да се раздели на няколко независими подзадачи.
  - Паралелни алгоритми може да се изпълняват и на еднопроцесорни компютри, което води до намаляване на продуктивността.
- **Разпределени** – при тях задачите и данните се разпределят на множество машини в мрежови и облачни среди.
- Серийните алгоритми са по-лесни за разбиране и реализация, а паралелните и разпределените изискват допълнително управление и синхронизация.

Сложност на алгоритъм

- **Сложност на алгоритъм** е приблизителна оценка O на максималния брой операции/стъпки, които са необходими за изпълнението на алгоритъм, в зависимост от броя на входните данни N.
- **Означава се с главно O, а в скоби след него се задава приблизителния порядък на броя стъпки,** напр. $O(N), O(N^2), O(N^3)$.
  - Сложността е функция на N, която може да се опише чрез конкретна формула, напр. $N^2 + N/2 + 5$.
  - В общия случай обаче е важен само порядъка („най-тежката" част от формулата), а не точната формула. За дадената формула отчитаме сложност $O(N^2)$.
- Броят стъпки е относителен, защото не може лесно да се прецени колко стъпки (описани в ЕП) на колко машинни инструкции точно съответстват.
- Трябва да се опитваме да търсим и реализираме алгоритми с по-малка сложност и съответно по-висока ефективност.

Видове алгоритми според сложността (1)

- **С константна сложност** O(1). Броят на операциите не зависи от броя на входните данни.
  - Пример: Изчисление на сумата на числата от 1 до N (които са N на брой), може да се реализира с формула за аритметична прогресия. (Ако не се сетим за формулата може и по-сложен алгоритъм да измислим.)
- **Линейна O(N).** Броят на операциите зависи линейно от броя на входните данни.
  - Пример: При търсене в списък от N елемента се обхождат всички и се проверяват един по един.
- **Квадратична** O(N^2). При обем на входните данни N, се изпълняват N^2 стъпки.
  - Пример: ако за два списъка с числа от по M елемента (тогава N=2*M), желаем да намерим произведенията на всеки елемент от единия списък с всеки елемент от другия списък, ще са необходими $M^2 = (N/2)^2$ броя основни операции.

Видове алгоритми според сложността (2)

- **Логаритмична** O(log(N)), O(N*log(N)). Това е относително ниска (между константна и квадратична) степен на сложност на алгоритъма, при която броят на входните данни N е равен на степен на броя на операциите.
- **Експоненциална** сложност има, когато броят на основните операциите се увеличава стремително при увеличение на броя на входните данни. Такива зависимости са $O(N^K)$, $O(K^N)$, O(N!) и изглеждат плашещи при по-големи стойности на N и K. (Квадратичната сложност е не толкова страшен случай на N^K.)
- **Сложност, зависеща от няколко променливи** – ако имаме два списъка с различен брой M и N на елементите, то всевъзможните двойки комбинации от елементи от двата списъка е M*N. Тогава сложността на алгоритъм, извеждащ всевъзможните двойки е O(M*N).

Програма и подпрограми

- Програмата описва основен алгоритъм.
- Подпрограмите описва подалгоритми, които могат многократно да бъдат използвани в основната програма.
- Подпрограмата е добре да имат самостоятелна логика, която зависи само от входни за нея параметри.
  - Не добър, в процедурното програмиране, вариант е подпрограмата да използва глобални параметри като входни данни. В този случай логиката ѝ става зависима от глобални данни, които не е известно как се контролират.
- Аналогично на функция в математиката програма/подпрограма може да връща резултат (от различни типове данни – число, низ…).
- В една подпрограма може да се използват други подпрограми.

Използване на подпрограма като „черна кутия“

- Подпрограмата може да се използва като „черна кутия“ от програмата. т.е. за основната програма са важни:
  - какво прави подпрограмата (каква е логиката ѝ);
  - какви входни данни трябва да получи подпрограмата;
  - какъв резултат се получава (тип на данните или отпечатването им, в конзолата, например).
- За основната програма не е важно как е реализирана подпрограмата.
- В една подпрограма може да се използват други подпрограми.

Видове подпрограми: процедури и функции

- **Функция – подпрограма, която връща резултат от изпълнението си.**
  - Резултатът е от някакъв основен (примитивен) тип – число, низ (текст), булева стойност (да/не, истина/лъжа, true/false, 1/0) – или по-сложен тип.
- **Процедура – подпрограма, която не връща резултат от изпълнението си.**
  - За да се знае какво се е случило при изпълнението, в този случай, подпрограмата трябва да изведе информация за потребителя в някакъв вид – например, текст в конзолата или да промени някакви глобални параметри.
- Процедура е частен случай на функция: **процедурата е функция, която връща нищо**.
- **(Затова) в C++ подпрограмите винаги се наричат функции.**

Основни действия с подпрограми в ЕП
(важна терминология)

- **Декларация** – задава се име, типове входни данни, тип на изходен резултат;
- **Дефиниция** – описва се алгоритъма, който се изпълнява върху входните данни;
- **Използване/Обръщение/Извикване** – обръщаме се към подпрограмата чрез името ѝ и конкретни входни данни.

Декларация на процедури и функции в C++

- **Декларацията описва заглавна част на подпрограма**. Всяка декларация си има
  - ***име*** – което ние измисляме (и което отговаря на определени правила);
  - ***входни параметри*** – не са задължителни (някои подпрограми може да не изискват входни параметри);
  - ***тип на връщана стойност***.
- **Декларация на процедура:**

  void име(списък_с_входни_параметри)

  „void“ означава, че подпрограмата не връща резултат
- **Декларация на функция:**

  тип_на_връщана_стойност име(списък_с_входни_параметри)

Примери за декларации на подпрограми в C++

- **void programInfo()** – декларация на процедура без параметри – очакваме (като гледаме името programInfo) да се изведе (в конзолата) някаква информация за програмата.
- **int sum(int a, int b)** – декларация на функция с два параметъра a и b (цели числа от тип int), която връща като резултат цяло число. Ще очакваме като резултат от изпълнението да се получава сумата на числата a и b.
- Параметрите описани в декларацията се наричат **формални параметри** – не са конкретни стойности, а променливи, които може да получават различни стойности при различни извиквания на функцията.

Декларация на програма в C++

- **Програмата е специална функция, чието име е „main“ и типът на връщана стойност е int:**

  int main()
- **Върнатата стойност указва как приключва програмата**:
  - 0 – успешно;
  - 1 (или друго различно от 0 число) – неуспешно.
- Върнатата стойност се използва от системата, извикала програмата В стандарта C++ не се указва как системата обработва върнатия резултат Не се указва и какви възможни интерпретации може да имат различни ненулеви стойности.
- Често срещан, но некоректен (неописан в стандарта) начин за декларация на основната програма е като void main().
- **Изпълнението на програма започва винаги от main-функция.**

Дефиниция на функция (процедура)

- **В дефиницията се описва алгоритъм със средствата на ЕП.**
- Дефиницията се огражда във фигурни скоби – лява { и дясна }, съответно за начало и край. Нарича се **тяло** или **блок на функцията**.
- Съдържа команди на ЕП, включително и извиквания на други функции.

Дефиниция на programInfo()

```
void programInfo()
{
	cout << "Sum of numbers\n";
}
```

- **Със „cout <<“ в конзолата се извежда низ** (текст описан в кавички) **или данни от друг тип**.
- cout е стандартен изходен поток, към който с оператора „<<“ може да предаваме данни
- Може последователно да изведем (да запишем) в потока множество разнотипни данни:

```
	cout << a << " + " << b;
```

Дефиниция на int sum(int a, int b)

```
int sum(int a, int b)
{
    return (a + b);
}
```

- Връщането на стойност става с команда return след, която се задава стойността (получена от изчислението на израз в примера)
- След return не може да се задават други команди.

Извикване на функции

- **Извикването** на дефинирана вече функция става чрез **името ѝ и списък от конкретни (фактически) параметри**, съответни на формалните.
- При това
  - **всеки формален параметър получава като стойност, съответния фактически параметър** и
  - **тялото на функцията се изпълнява за конкретизираните вече формални параметри**.

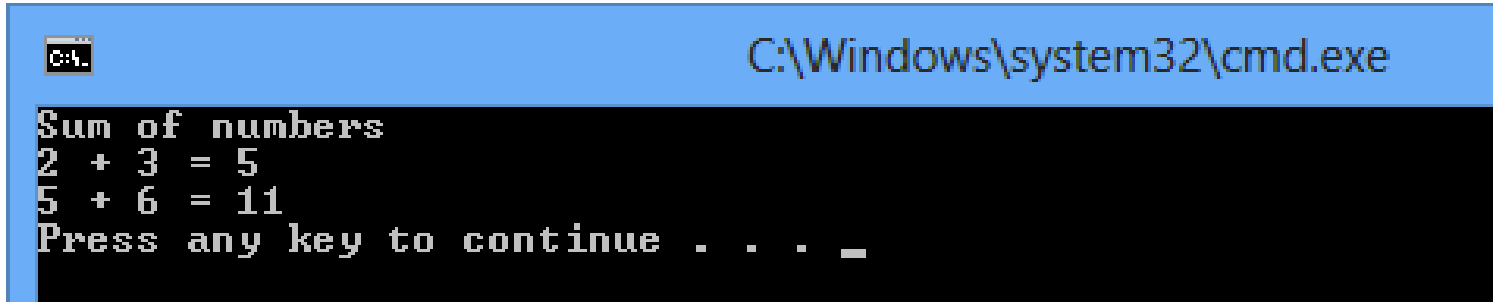
Дефиниране на main-функцията

```
int main()
{
    programInfo(); // извикване на programInfo()
    // отпечатване в конзолата на низ, резулатата от
    // извикването на sum(2, 3) и символ за край на ред "\n"
    cout << "2 + 3 = " << sum(2, 3) << "\n";
    // отпечатване в конзолата на низ, резулатата от
    // извикването на sum(5, 6) и символ за край на ред "\n"
    cout << "5 + 6 = " << sum(5, 6) << "\n";
    return 0; // успешно приключване на програмата
}
```

- Функцията sum сме извикали двукратно, с различни фактически параметри.
- При извикването на подпрограма тя започва да се изпълнява…
- …а след изпълнението на подпрограмата, продължава да се изпълнява следващата след нея команда .

Цялостен пример

```
#include <iostream>
using namespace std;
void programInfo() {
        cout << "Sum of numbers\n";
}
int sum(int a, int b){
        return (a + b);
}
int main(){
    programInfo(); // извикване на programInfo()
    // отпечатване в конзолата на низ, резулатата от извикването на sum(2, 3) и символ за край на ред "\n"
    cout << "2 + 3 = " << sum(2, 3) << "\n";
    // отпечатване в конзолата на низ, резулатата от извикването на sum(5, 6) и символ за край на ред "\n"
    cout << "5 + 6 = " << sum(5, 6) << "\n";
    return 0; // успешно приключване на програмата
}
```



- Чрез `#include <iostream>` се включва стандартната библиотека iostream за вход и изход.
- Обектът-поток cout е дефиниран и достъпен в областта на имената (namespace) std, описана в библиотека iostream .

Задача: Всяко повторение на еднотипен код може да доведе до

- възникването на грешки (поради грешно изписване) или
- проблеми при последваща необходимост от промени на кода на няколко места (и съответно пропускането на някое от местата).

Оптимизирайте main-функцията, така че да не се повтаря кода за извеждане на резултатите!